Typing heterogeneous dataflow graphs for static buffering and scheduling

Pierre Donat-Bouillud
Sorbonne Universités/STMS/Inria
pierre.donat-bouillud@ircam.fr

Jean-Louis Giavitto
CNRS/Sorbonne Universités/STMS
jean-louis.giavitto@ircam.fr

ABSTRACT

Interactive multimedia system usually represents computations as a dataflow graph – the patching model. Nowadays, dataflow graphs integrate multiple medias from various dedicated languages and systems, with multiple rates, for instance audio rate, video rate, control rate, but also the rate of FFT frames, as well as requirements on buffer sizes, or on what kind of data they consume or produce. We design a type system for the DSP extension of Antescofo that makes it possible to safely combine effects in such a heterogeneous graph. We show how to infer the types given known types in the graph, and how we use this type system to compute buffer sizes and to schedule the graph. Our approach is exemplified on an Antescofo program that does speed tracking of a foreground object in a video stream to drive an audio effect.

1. INTRODUCTION

The patching paradigm is nearly universal in modern computer music real-time environments. It corresponds to a periodic dataflow model where the fundamental unit of time, the tick, is defined by n audio samples.

This computation model has also been proven relevant and useful for the processing of other kind of signals, from video to sensor data streams like accelerometers or GPS. The difference lies in the kind of samples (a whole image in video, or a few floats for an accelerometer) and in the data rate.

New interactive multimedia applications require more and more the integrated management of such streams. This problem has been addressed in *ad-hoc* ways in current environments. For example, GEM is a set of external objects for PD where images (GEM states) are referred through a handle. Handles are transferred in the control graph from a processing node to the other. A node receiving an handle can access the image through the handle and creates a new handle to refer to the output result.

This approach is effective but suffers from several drawbacks. Such handling of non-audio signal lives on the control level. Thus the fact that the computations to perform on it are periodic, is not taken into account. For example, one may imagine that the strict succession of signal samples is not granted if samples do not take the same computation paths. Another side effect is that the management of different kinds of signals becomes asynchronous: the temporal alignment of signal (*e.g.* audio and video) must be reconstructed, for example by time-stamping each samples.

A natural solution to avoid these problems is to unify the processing of the various kinds of signals under the same dataflow graph where each node defines a signal transformation. However, the computation model must be extended to handle different sample types and different rates within the same graph.

Multirate signal processing is usually considered through up- and down-sampling operations to deal with heterogeneous ADCs and DACs (sample-rate conversion), to minimize computational cost and in the development of complex filter (*e.g.* to minimize noise power). Here multirate is tackled so as to unify the management of heterogeneous signals in the same dataflow graph to ensure synchrony.

Contributions. In this work, we address the handling of multirate signal processing by developing a *type inference system*. Our objective is to infer for each signal processing node, the relevant rate and the type of data consumed and produced at each periodic node activation. A link between a source and a target node is seen as an active *adapter* that uses a buffer to adapt the production rate of the source node to the consumption rate of the target node.

The type of nodes and links is used to derive the length of the communication buffers and to manage the scheduling of computations through the determination of the period of activation for each node and link. The type system accommodates generic processing nodes corresponding to the parametric implementation of a signal transformation (the parameter being the buffer length). For instance, such parametric implementation are produced as an output of the Faust compiler.

This type system is used to extend the patching sublanguage of Antescofo [1] to the handling of heterogeneous dataflow graphs. Antescofo patches are textual description of synchronous dataflow graph where the nodes are defined as external plugins or in an external dedicated language (like Faust).

The type inference approach proposed here is particularly relevant to the Antescofo framework. It makes it possible to set scheduling constraints on given nodes (like input and output nodes), it determines the instance of a parametric node that minimizes the memory footprint and is fully independent of the actual implementation of the processing node once an adequate type is specified for them.

Organization of the paper. Next section gives some background elements of the Antescofo system and the patching sublanguage. The types associated to a DSP node and to

a DSP link are described in sect. 3 with the inference procedure. The information captured by the type system are used to produce a static scheduling of the signal computations, in sect. 4. An application, sect. 5 combining audio and video signals illustrates our approach.

2. EMBEDDING DSP EFFECTS IN ANTESCOFO

Antescofo [2] is a score following system that combines a listening machine with a reactive engine. It uses a dedicated synchronous language to define an *augmented score* where musical events to follow, the electronic reactions to these musical events, and the synchronization between these actions and the human performers, are specified together. Its has been primarily used for written mixed music, such as for *Anthèmes 2* (1997) by Pierre Boulez. Antescofo can drive sound synthesis, but also video displays, light scenarios, and also react to some arbitrary analysis of the audio signal or to the speed of an accelerometer. Antescofo is usually embedded in a host such as Max/MSP [3] or Puredata [4], that perform all the signal processing while Antescofo is in charge of the control computations and of the timing mechanisms. However, an Antescofo extension for signal processing [1] has started to be developed.

In *Antescofo*, signals flow through processing nodes, called *effects*, which transform samples, connected through links that can modify the temporal characteristics of the signal.

2.1 Instantiating an effect

Effects and *links* are declared in the score and are instantiated at parsing time, whereas the connections between *effects* can be changed all along the performance. The declaration

```
@dsp_def my_effect : type := dsp::F(arg1, ..., argn)
```

introduces an instance my_effect of a dsp node dsp::F with the optional type specification type. Giving an explicit type to a dsp node makes possible to bypass the type inference systems, for instance to impose some constraints induced by the environment. The argument in the right hand side are instantiation parameters (*e.g.* the size of a FFT window). The effect dsp::F can be a builtin effect or can be defined in another DSP processing language, such as Faust [1], for which effect can be defined with a @faust_def.

Links are declared only to specify the identifiers that can be used in a patch:

```
@dsp_channel $$my_channel
```

2.2 Connecting effects

Effects are connected together to create a dataflow graph, that typically takes an audio signal from the soundcard or the host environment, and sends back a transformed signal. In Antescofo, connecting *effects* is an elementary action in the score, called a *patch* action. Patches describe the dataflow graph in a functional style: it lists a number of equations with the outputs on the left-hand side, and the digital signal processor and its inputs on the right-hand side.

In Fig. 1, a builtin sampler that plays a wav sound file is connected to the audio output. The type specifies that the sampler takes one control input (to trigger the playback) and outputs two data: a signal (from the sound file) and a control value that indicates the end of the playback. The whenever construction defines a reaction which is performed each time its condition $end_sample is set to true. The boolean control variable $play_sample triggers the playback. Notice the intersection between the control variable in the program and the control variable in the patch. The patch plugs the sampler through control variables and links. When $play_sample is set to true, the sampler starts its playback. Once the playback finished, the output $end_sample is set to true by the effect, which triggers the reaction, making it easy to loop a sample for instance.

As an Antescofo action, a patch action can be played after detecting some musical event, waiting for some delay, and can be synchronized with the usual synchronization strategies [5] of Antescofo.

3. THE TYPE SYSTEM

The previous example gives a glimpse of Antescofo DSP types. Languages such as Kronos [6] or Faust [7] have introduced a sophisticated type system, including support for multirate signals, to describe signal processing low-level dataflow operations, while Antescofo type effects are seen as blackboxes, that can be coded in very different languages, such as C++, or Faust. Typing these blackboxes adds some information that makes it easier to combine them correctly and schedule them.

The Antescofo signal type system aims at grasping the following idea: a signal is a timed sequence of samples. For efficiency reason, samples are not handled one by one, but periodically in group of a fixed size s. But if a DSP node processes n samples every p seconds, nothing (except latency) prevents from scheduling the DSP computation two times every $2p$ seconds to process $2n$ samples.

```
$play_sample := false
$end_sample := false

@dsp_channel  $$out
@dsp_def dsp::my_sampler
: $ -[88200]⟶ $$, $
:= dsp::sampler("sample.wav")

whenever($end_sample) { print "Playing Done" }

patch{
$$out, $end_sample := dsp::my_sampler($play_sample)
dsp::output[0]($$out)
}
; ...
$play_sample := true
```

Figure 1. An Antescofo score where a sampler is connected to the soundcard output. The sampler used a 88200 samplerate, has one scalar input to indicate when the sample must be played, one audio signal output and one scalar output to say when the sample has been played. The syntax of type expressions is presented in sect. 3

3.1 The language of types

We define the primitive types of our signal type system as (n_i are constant integers):

$$\begin{aligned}\text{PTYPE} ::= &\ \text{int} \mid \text{float} \mid \text{double} \mid \text{PTYPE} \times \text{PTYPE} \\ &\mid \text{Array}(\ \text{PTYPE}, n_1, n_2, \dots)\end{aligned}$$

The construction $\text{Array}(\ \text{PTYPE}, n_1, \dots)$ is a dependent type where n_i represents the number of element in the ith dimension of a multidimensional array.

The type of a signal is a term:

$$\begin{aligned}\text{STYPE} ::= &\ \$ \mid \text{VAR} \mid \text{STYPE} \times \text{STYPE} \\ &\mid \text{Signal}(\ \text{PARAM},\ \text{PARAM},\ \text{PTYPE})\end{aligned}$$

The symbol $ denotes the type of a control (a scalar value) which can be seen as a "sporadic signal": a succession of values without constraint of periodicity. The set VAR is the set of type variables $\alpha, \beta, \dots$ that makes it possible to have generic types. Given a name to a yet-to-be-inferred type makes it possible to refer to this type elsewhere (*e.g.* to constraint behavior of some nodes). If the variable must not be referred elsewhere, one can use $$ to spare an identifier.

The set PARAM is the set $\mathbb{N} \cup Id$ of signal parameters. Signal parameters are either an integer constant or a variable whose value must be deduced by the type inference algorithm. We use the letter f and s to refer respectively to the first and the second element of the tuple. In $\text{Signal}(f, s, e)$, f represents a frequency in Hertz, s is a buffer size, and e is the type of each element of the stream.

Effects and *links* are seen as functions that transform *signals*, that's to say they have the following types:

$$\text{FTYPE} ::= \text{STYPE} \rightarrow \text{STYPE}$$

3.2 Type Constraints

Effects are functions on signals or tuples of signals where all incoming and outgoing signals are constrained to the same frequency whereas *links* do not touch the samples but reorganizes the streams in term of succession. That is to say, they can change the frequency of a stream and its buffer size as long as the "bandwidth" remains the same. *Links* should not modify samples: for instance, they cannot be used to mix stereo channels into mono, which should be performed by a dedicated node.

For example, a DSP effect e consuming p signals and producing q signals has type:

$$\begin{aligned}e : &\ \text{Signal}(f_1, s_1,\ \mathrm{t}_1) \times \dots \times\ \text{Signal}(f_p, s_p,\ \mathrm{t}_p) \rightarrow \\ &\ \text{Signal}(f'_1, s'_1,\ \mathrm{t}'_1) \times \dots \times\ \text{Signal}(f'_q, s'_q,\ \mathrm{t}'_q)\end{aligned}$$

where $f_1 = \dots = f_p = f'_1 = \dots = f'_q$. For the sake of brievety, as $f_1 = \dots = f_p = f'_1 = \dots = f'_q$, the common frequency can be specified by annotating the arrow:

$$\begin{aligned}e : &\ \text{Signal}(s_1,\ \mathrm{t}_1) \times \dots \times\ \text{Signal}(s_p,\ \mathrm{t}_p) \text{—}[f] \rightarrow \\ &\ \text{Signal}(s'_1,\ \mathrm{t}'_1) \times \dots \times\ \text{Signal}(s'_q,\ \mathrm{t}'_q)\end{aligned}$$

A link e has type:

$$\begin{aligned}e : &\ \text{Signal}(f_1, s_1,\ \mathrm{t}_1) \rightarrow \\ &\ \text{Signal}(f'_1, s'_1,\ \mathrm{t}'_1) \times \dots \times\ \text{Signal}(f'_p, s'_p,\ \mathrm{t}'_p)\end{aligned}$$

where $\mathrm{t}_1 = \mathrm{t}'_1 = \dots = \mathrm{t}'_p$. The target of a link is described by several signal types because it may appears has the input of several DSP node, each imposing its own constraint. If a link identifier occurs p times in the right hand side of the patch equations, then the size of the tuple of types in the right hand side of the arrow is p.

We also enforce a constraint on links type that represents the rate of consumption over production in the link:

$$\forall i \in [\![1, p]\!],\ \frac{s_1}{f_1} = \frac{s'_i}{f'_i} \tag{1}$$

It means that if s samples are produced at frequency f, then in a period $\frac{1}{f}$, $\frac{s}{f}$ are produced. The *link* must output the same number of samples in one output period, which entails the above equality.

The patch action finally defines a *DSP graph*. A *DSP graph* is also a function on signals, resulting of alternatively composing *effects* and *links*: *links* connect *effects*, with inputs and outputs which are effects. Usually, most effects in a *DSP graph* will have generic types, except inputs and outputs to the signal sources and sinks, and embedded legacy effects.

3.3 Type inference, or solving the constraints

Antescofo embeds Faust effects and builtin effects. Each effect is associated to a type T and each instance of an effect has an associated type derived from T. Many effects are *generic* (or *polymorphic*), meaning their inputs and/or outputs are parametrized in one or several of the three components of the type of a signal. For instance, Faust effects are sets of equations on samples and the Faust compiler produce a `compute` function that is parametrized by the length of the processed buffers (and so accept any buffer size).

Inferring the types of a *DSP graph* entails inferring the types of *links* that are always generic, as well as to infer the *generic* types of the effects, and checking if the already defined types are coherent. Consequently, the inference of types has to be done for each DSP graph definition, *i.e.* for each patch action. Note that in a working DSP graph, *i.e.* a graph that has some outputs, there is at least one node that has a *non-generic* type (the output).

The idea of the type inference algorithm is to propagate the known types in the graph to nodes with generic types, until all types are known, by looking for a *fixpoint*. We do not show here all the propagation rules (or *inference rules*) as it would be beyond the scope of this article.

Fig. 2 shows a *link* that connects Effect1 to Effect2. The *link* has a generic input type α and a generic output type β. Different propagation rules can arise depending on what component of a Signal are generic. Here, we suppose that we know frequencies f_1 and f_2.

- If s_2 is known, we can compute s_1 using Eq. 1. If s_1 is also already known, Eq. 1 has to hold.
- If s_1 is known but not s_2, we also apply Eq. 1.

We also have to have $t_1 = t_2$ as a *link* does not manipulate the samples themselves.

Fig. 3 shows how known types on *links* are propagated to an *effect*. A constraint on frequencies is that all the frequencies of the types on Effect must be the same frequency

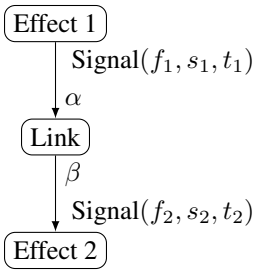

Figure 2. Propagation of types from *effects* to *links*.

f. We choose $f = \max(f_1, \ldots, f_n)$, which means that the effect must run at the rate of its quickest inputs and outputs. This strategy minimizes the required buffer length. Another strategy is to choose $f = \min(f_1, \ldots, f_n)$ to minimize the number of computations (at the expense of computation length).

Propagations are iterated until we reach a fixed point, that's to say all frequencies, buffer sizes, and element types are known and do not change on two consecutive iterations. The number of iterations cannot exceed the diameter of the graph which is well defined because the DSP graph is acyclic by construction. The propagation of *frequency* and *buffer size* must be done at the same step, as they are linked by Eq. 1. Then, we do the PTYPE inference and checking.

On Fig. 4, two effects Effect1 and Effect2 with generic types $\beta, \beta', \gamma, \gamma'$ are connected together and are connected to links that have known types, *i.e.* Signal(f, s, e) and Signal(f', s', e'). If we suppose that these known types are different, the inferred types for Effect1 and Effect2 could be different depending on where we start propagating types. In the current implementation, types are first propagated from the outputs, but we could choose a propagation order to optimize some function on the whole DSP graph, for instance to maximize buffer sizes to optimize for performance with vectorization, or to minimize buffer sizes to optimize for temporal precision.

4. SCHEDULING

During the performance, the execution of the dsp graph is driven by a period called *dsp tick*. Every dsp tick, some nodes are activated, they consume some data available in the buffer's links, do some computations, and produce some data in the buffer's links.

Each time a patch action is found, the dsp graph is modified: first, we solve the type constraints for the new graph, and the scheduling order is computed to take into account the dependencies of the effects, with a topological sort. After that, the types of the nodes are used to compute their actual periods of activation. For an *effect*, *inputs* and *outputs* have the same period, wich is this activation period.

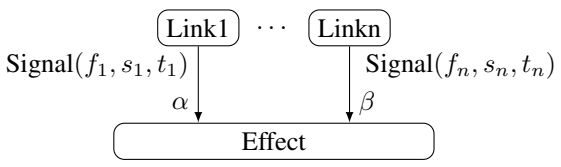

Figure 3. Propagation of types from *links* to *effects*

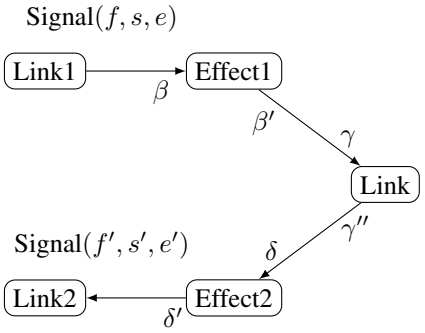

Figure 4. Chained generic effects.

For *links*, which convert the "*impedance*" of signals and as such do not have the same periods in their inputs and outputs, the activation period is the period of the input type.

The dsp tick is computed as the smallest tick such that it divides all the periods of all the nodes, that is to say, their greatest common divisor (GCD).

Links. *Links* store an internal circular buffer that is used to adapt to the various rates. As *links* do not perform mixing, it has only one input, but they can have several outputs. The *effect* that is connected to its input writes in the internal buffer and the ones that are connected to its outputs read it. We use virtual memory functionalities (*mmap* system call on Linux and mac OS) to remap the memory addresses after the end of the buffer into the the buffer itself. It makes it possible to directly give the effect a pointer to the internal buffer, without having to copy buffers that would span the end and the beginning of the circular buffer, thus optimizing for less copying. It also means that we can allocate memory only multiple of a page size, typically, 4 KiB on x86 processors. For a graph with 10 effects and 20 links, the memory consumption will be roughly 80 KiB which is quite small for modern computers, and even for small boards such as the Raspberry Pi. [1]

Control. An effect processes one buffer of the size indicated by its type at each tick but can modify or read an antescofo variable at buffer boundary.

5. TWO DIFFERENT RATES AT PLAY: VIDEO AND AUDIO

The type system is flexible enough to accomodate very different rates and effects. We developed a proof of concept that does speed tracking of the largest foreground object in a video, to control an audio effect. It can be used to roughly track the speed of a waving arm, for instance. The video input has typically a rate of an order of magnitude of 10 Hz, for example, 29.97 frames per second, whereas the audio output usually requires a 44.1 kHz samplerate to keep all human-perceivable frequencies. Video frames and audio rates must also be carried in the same way through the DSP graph.

In Puredata [4] with Gem [8], although Puredata makes it possible to change the samplerate in a subpatch using a `block` object, it is difficult to have several rates live together in the same patch, as mixing video and audio would

[1] The Raspberry Pi 3 has 1 GiB RAM.

```
BPM 120

$speed := 0.
$max_speed := 15
$pitch_freq := 0;
$c0 := 16.35
$c7 := 2093.00

@faust_def faust::SimpleSynth($frequency)
{
    import("stdfaust.lib");

    freq = hslider("frequency", 16.35, 16.35, 2093.0,
        0.01) : si.smoo ;

    process = os.osc(freq)  : re.mono_freeverb
        (0.5,0.5,0.5,23);
}

@dsp_def dsp::webcam := dsp::camera(0)
@dsp_def dsp::tracking := dsp::speedtracking()

@dsp_def dsp::synth := dsp::SimpleSynth()

@dsp_def dsp::audioOut := dsp::output(0)

@dsp_channel $$video
@dsp_channel $$out

whenever($speed)
{
    print "Speed update"
    $pitch_freq := $c0 + @min($max_speed, $speed) * (
        $c7 - $c0) / $max_speed
}
6 print Start

patch{
    $$video := dsp::webcam()
    $speed := dsp::tracking($$video)
    $$out:= dsp::synth($pitch_freq)
    := dsp::audioOut($$out)
}
40s print DONE DONE
```

Figure 5. An Antescofo score that uses speed tracking of an arm to control a synthesiser.

require. In GEM, a `gemHead` object creates [9] a state that can store images, and a pointer to this state is carried through the inlets and oulets in a Puredata `atom`, as a `gemList`, *i.e.*, the frames are not carried as signals. In Chuck [10], results of unit analyzers are stored in an object called a `UAnaBlob` [11] which contains a timestamp indicated when it was computed, whereas in Antescofo, spectral bins resulting from a FFT for instance would also be represented as a signal, but with a different rate depending on the parameters of the FFT.

A type system ensures that frames can be carried safely and in a general way within the DSP graph: a video stream with a framerate fps, seen as a stream of images of given width and height, will have a type such as

$$\text{Signal}(fps, 1, \text{Image}(\texttt{width}, \texttt{height}))$$

as shown on Fig. 6. $\text{Image}(\texttt{width}, \texttt{height})$ is an alias for $\text{Array}(\text{int} \times \text{int} \times \text{int}, \texttt{width}, \texttt{height})$. The output of the speed tracking node is a control variable, that is updated for each frame. It means that we can further process

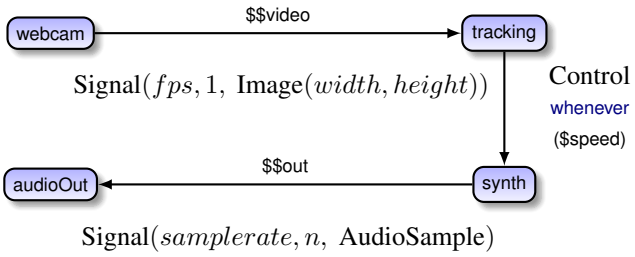

Figure 6. The DSP graph is made of four main nodes: a input node connected to a video source (video camera or video file), a node that does speed tracking, a node that plays a sound, and an audio output, to the soundcard.

this control variable, by detecting when it changes with a whenever, as shown on Code 5. In Antescofo, the whenever control instruction watches a condition on variables of the score and computes something when the values of the variables changes and the condition evaluates to true. The code associated to that whenever computes here a frequency from the speed. After that, the frequency is used to drive a synthesizer which is coded in Faust.

Speed tracking To track the speed of a foreground object, we embedded the library OpenCV [12] in Antescofo. The speedtracking effect is a builtin effect coded in OpenCV. We extract the foreground using the Substractor Background MOG2 [13], eroding and deleting the result to get rid of noise, and then detecting the contours and keeping the largest one with respect to its area, as shown on Fig. 7. The speed is computed by measuring the displacement of its mass center, and smoothed. When the detected contour changes are higher than a given threshold, the speed is reset.

Synthesizer The Faust effect is embedded in Antescofo as described in [1]. The input frequency is smoothed then used to drive a simple oscillator, to which we add some reverb using freeverb, an opensource implementation of a Schroeder/Moorer reverb model [14].

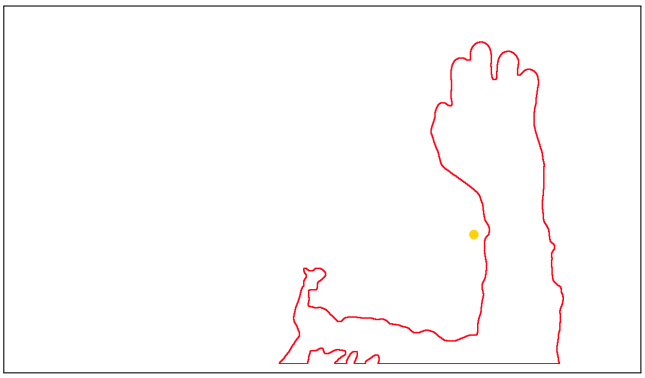

Figure 7. Detection of waving arm and hand in a video using OpenCV. The centroid of the contour is the yellow point left to the wrist.

6. CONCLUSIONS

We have defined a type system for the dsp extension of Antescofo. The type system makes it possible to combine heterogeneous nodes that operate on streams with various rates, and various types of elements. The type system is

used to check for the correctness of the dsp graph, to statically allocate the communication buffers between effects, and to schedule them. Most nodes have generic types, for which the actual types are inferred given already known types, which can enable Antescofo to choose an inference strategy that can optimize some performance or precision metrics. The flexibility of the type system is showcased on an example that connects various heterogeneous effects, *i.e.* a Faust effect, an OpenCV effect, and controlled by Antescofo code.

We aim at allowing even richer types, that can express more complex constraints, such as affine relations between frequencies and buffer sizes among several inputs and outputs of a node, to more precisely state how effects can be connected together. For instance, it would be useful to express that the buffer size of one input must be the double of another input on an effect.

In this article, we have described how the type system makes it possible to *statically* choose a buffering size and schedule the effects. However, depending on the current conditions and requirements of the execution of the score, that is to say, the overloading of the processor, and the need for more temporal precision, the type system could express how the frequency and buffer sizes of the dsp graph can be adapted individually to react *dynamically* to them, for instance by specifying ranges. To improve accuracy in the handling of control, we envision to use dynamic buffer sizse when the processing node accept generic size. At the occurrence of a control, the buffer current buffer processing is stopped and the input buffer are properly split in two (corresponding to the data before the occurrence of the contro and the date after this occurence). Then these buffers are propagated into the rest of the dsp graph. The motivation is to achieve almost sample accuracy while keeping the buffers as large as possible, to benefit from vector instructions and cache locality.

Acknowledgments

We would like to thank Clément Poncelet for proofreading the article, and Florent Jacquemard for his support.

7. REFERENCES

[1] P. Donat-Bouillud, J.-L. Giavitto, A. Cont, N. Schmidt, and Y. Orlarey, "Embedding native audio-processing in a score following system with quasi sample accuracy," in *ICMC 2016-42th International Computer Music Conference*, 2016.

[2] A. Cont, "Antescofo: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music," in *Proceedings of International Computer Music Conference (ICMC)*, Belfast, Irlande du Nord, August 2008.

[3] D. Zicarelli, "How I learned to love a program that does nothing," *Computer Music Journal*, vol. 26, no. 4, pp. 44–51, 2002.

[4] M. Puckette *et al.*, "Pure Data: another integrated computer music environment," *Proceedings of the second intercollege computer music concerts*, pp. 37–41, 1996.

[5] A. Cont, J. Echeveste, J.-L. Giavitto, and F. Jacquemard, "Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo," in *Proceedings of International Computer Music Conference (ICMC)*. Ljubljana, Slovenia: IRZU - the Institute for Sonic Arts Research, Sep. 2012. [Online]. Available: http://hal.inria.fr/hal-00718854

[6] V. Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, 2016.

[7] Y. Orlarey and P. Jouvelot, "Signal rate inference for multi-dimensional faust," in *The 28th symposium on Implementation and Application of Functional Languages (IFL 2016)*, 2016.

[8] M. Danks, "Real-time Image and Video Processing in GEM." in *ICMC*, 1997.

[9] J. Zmölnig, "Gem for pd-recent progress." in *ICMC*, 2004.

[10] G. Wang, P. R. Cook, and S. Salazar, "Chuck: A strongly timed computer music language," *Computer Music Journal*, 2016.

[11] G. Wang, R. Fiebrink, and P. R. Cook, "Combining Analysis and synthesis in the Chuck Programming Language." in *ICMC*, 2007.

[12] G. Bradski, "OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[13] Z. Zivkovic and F. Van Der Heijden, "Efficient adaptive density estimation per image pixel for the task of background subtraction," *Pattern recognition letters*, vol. 27, no. 7, pp. 773–780, 2006.

[14] M. R. Schroeder, "Digital simulation of sound transmission in reverberant spaces," *The Journal of the Acoustical Society of America*, vol. 47, no. 2A, pp. 424–431, 1970.